System and method of implementing a customizable software platform.

DESCRIPTION

CROSS-REFERENCES TO RELATED APPLICATIONS

[Para 1] This application claims the benefit of U.S. Provisional Application Serial No. 60/481444, filed on September 30, 2003, entitled "A method of building a customizable software platform, for the management and development of service-oriented software modules, upon itself", which is incorporated herein by reference.

BACKGROUND OF THE INVENTION

[Para 2] 1. Field of the Invention

[Para 3] The present invention relates to the field of computer software and, in particular, to a system and method of building a software platform that is used for management and development of service-oriented software modules and applications; this system and method leverages its own service-oriented artifacts in its own construction.

[Para 4] 2. Description of the Related Art

[Para 5] Relational Database Management Systems (RDBMS) provide an example of a system that uses its own core artifacts as part of its own construction. For example, in most RDBMS, "system" tables are used to hold catalog and metadata information about all end user and system tables. The idea is that if database tables are a good way to hold data and metadata, the

RDBMS itself should use it to hold its own information. Indeed, by using the system artifacts as part of the implementation of the system, the system inherits some of the benefits that it aims to provide the end user.

[Para 6] The current art of building software platforms that aim to provide service-oriented development and management capabilities do not utilize self-produced, service-oriented artifacts as a significant part of their construction. As a result, construction and customization of such platforms do not directly benefit from the advantages of service-oriented architecture and programming methods. The current art of implementing service-oriented development and/or management results in platforms that are too hard, impractical or even impossible to customize at a granular level.

SUMMARY OF THE INVENTION

[Para 7] A principle object of the present invention is to provide a method for constructing a customizable, service-oriented, software system that can be used for the management, development and assembly of software service modules including, but not limited to, "Web Services". A major distinguishing aspect of the method for constructing the software system is to use the functionality, frameworks and methods provided to the end-user of this software system, in constructing the functionality of the software system itself. In other words, portions of the software system are constructed upon the artifacts of the system and through the same tools that are provided for the end users of the system.

[Para 8] The present invention aims to leverage the end functionality of the software system to allow the users of the system to customize, replace, or extend the built-in features provided by the software system by applying the functionality, frameworks and methods provided to the end-users of the system. The present invention provides a method for the rapid convergence of quality of the software system under construction, while building and

advancing the software system, through the compound reuse and leverage of its end-user functionality and its artifacts in building the software system itself.

[Para 9] Furthermore, the present invention significantly reduces the time for constructing, advancing and customizing the software system through the application of a service-oriented architecture and method that directly leverages the end-user functionality and artifacts of the software system.

[Para 10] The present invention provides a service-oriented method of communication to accomplish transparent distribution for the parts of the system as consumers of software services from the parts of the system as producers of software services. Furthermore, this method of communication provides a service-oriented method of broadcasting system and application events across distributed nodes of the software system.

[Para 11] Other objects and advantages of this invention will be set in part in the description and in the drawings that follow and, in part, will be obvious from the description or may be learned by practice of the invention. Accordingly, the drawings and description are to be regarded as illustrative in nature, and not as restrictive.

[Para 12] To achieve the forgoing objectives and in accordance with the purpose of the invention as broadly described herein, the present invention provides methods, frameworks, and systems for building a software system for the management, implementation and composition of service-oriented, software modules that are built themselves on top of service-oriented software modules leveraging its own artifacts and set of functionality in management, implementation and composition of software modules. In preferred embodiments, this technique comprises: 1) dividing the system into a core module, the Kernel, that provides a basic framework utilizing software interfaces with plug-able implementations for the dispatch of software service module implementation with a multi-threaded process abstraction; 2) use of said framework by the Kernel itself and the management and design tools to provide the rest of the functionality that the system needs in performing its own tasks including, but not limited to, metadata management, logging, cache

management, system configuration, shared memory management, event broadcasting and notification, security and provisioning as a set of service modules dispatched by the Kernel.

[Para 13] The method further comprises a mechanism for the consumption of software modules, based on their interfaces, decoupled from the implementation module dispatched within the Kernel and appropriately abstracted to provide transparent distribution. The Graphical User Interface (GUI) portion of the system used for defining service interfaces, definition, composition and management tools, consumes "System" services to perform its function, while the implementation of the those services are dispatched by the Kernel and implemented through the same framework that is provided to the end-user for managing, implementing, deploying and developing any software service module. Each function for the system is first described using a set of service interface definitions using the system itself, while the GUI layer providing this function accesses the function through the consumption of software services based on the defined interfaces.

[Para 14] The present invention will now be described with reference to the following drawings, in which like reference numbers denote the same element throughout. It is intended that any other advantages and objects of the present invention that become apparent or obvious from the detailed description or illustrations contained herein are within the scope of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[Para 15] Figure 1. A software service interface.

[Para 16] Figure 2 - Example composite model implementation.

[Para 17] Figure 3 - Exploded Composite service.

[Para 18] Figure 4 - Service oriented application architecture.

[Para 19] Figure 5a - Log Analyzer -- Graphical User Interface calls composite services.

[Para 20] Figure 5b. Log Analyzer - get aggregate service statistics; an example composite service called by the GUI.

[Para 21] Figure 6. Services used for Metadata management.

[Para 22] Figure 7a. Search for Web Services GUI - calls System services.

[Para 23] Figure 7b. 'Search for Web services' GUI - using 'FindWebServices' composite service.

[Para 24] Figure 8. Logging service execution - kernel calls a customizable composite service.

[Para 25] Figure 9a. -- Services provided for Service Process Monitoring and management.

[Para 26] Figure 9b. - 'Web services task manger' GUI - calls Monitoring and Management Services.

[Para 27] Figure 10a. - Services provided for Service Cache management.

[Para 28] Figure 10b. - Web Service Cache Manager GUI - calls Cache Management services

[Para 29] Figure 11. - NextAxiom System Architecture.

TERMS AND DEFINITIONS

[Para 30] A *software service*, or service for short, including but not limited to a Web service, is a discrete software task that has a well-defined interface and may be accessible over the local and/or public computer networks or maybe only available on a single machine. Web services can be published, discovered, described, and accessed using standard-based protocols such as UDDI, WSDL, and SOAP/HTTP.

[Para 31] A *software service interface*, in concept, represents the *inputs* and *outputs* of a black-boxed software service as well as the *properties* of that service, such as name and location. Take, for example, the interface of a simple software service named GetStockQuote, which retrieves simple stock quote information [FIGURE 1]. This service takes a ticker symbol input and returns the last trade price amount as well as some additional stock quote details, such as the day high and day low. Note that in order to use, or *consume*, a service, only knowledge of its interface is required. This means that as long as the interface of a service remains the same, different implementations of the service can be swapped in and out without affecting its consumers. This, as well as the fact that a service is a language- and platform-neutral concept, is one of the keys to the flexibility of service-oriented architectures.

[Para 32] An *atomic service* is a software service that is implemented directly by a segment of software code. In the existing NextAxiom™ HyperService™ Platform, atomic Web services are dispatched via a library. A library is a light, language- and platform-neutral wrapper that is linked to one or more atomic Web service implementations. Atomic Web services are logically indivisible Web services that represent "raw materials" to the HyperService™ platform.

[Para 33] A *composite service* is a software service that consumes any number of other atomic or composite services. In the HyperService™ platform, a composite Web service is implemented with a metadata-driven model that is automatically interpreted by a high-performance run-time engine.

[Para 34] Visual metadata models, which represent composite software services implementations to the HyperService™ system, are created in a graphical, design-time environment and stored as XML models. This environment offers a new and powerful visual modeling paradigm that can be leveraged to enable the visual modeling of transactional behavior. This environment was specifically designed to enable collaborative, on-the-fly creation of software services by business process analysts or functional experts, who understand the business logic and application required to implement real-world business processes and applications, but have no

knowledge of programming paradigms or Web service protocols. FIGURE 2 captures the implementation of a composite software service named "Expedite 3000 Series". This service is used by a master planner to expedite 3000-series inventory items when they fall short on the shop floor. This service was developed collaboratively and reuses services that were selectively exposed by the Inventory and Purchasing departments to the developers of this service.

[Para 35] Any software service that is consumed by a composite service model is said to be "nested" or "embedded" within that composite service. FIGURE 3 depicts a hypothetical composite service that resides in Chicago. This software service is composed of other composite services that are distributed across the country.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

[Para 36] A service-oriented software development paradigm effectively encapsulates the graphical user interface (GUI) aspect of any software application from its application logic. Furthermore, the use of service-oriented standards provides an avenue for encapsulating and reusing the application logic from existing software assets such as databases, operating system functions, data files, application files, and existing IT applications. The pivot point of service-oriented encapsulation is the definition of the software service interfaces. The benefit of service-orientation can be summarized as increased architectural flexibility and customizability, quality, software component reuse, and a significant reduction in the time and cost required to develop software solutions.

[Para 37] The present invention discloses a method of implementing a system for dispatching service implementations and the management and composition of software service modules, that uses its dispatch, management, discovery and composition functionality provided to the end-users and the artifacts of the system in its own construction. Applying the present invention in the

construction of NextAxiom's Hyperservice platform (available at www.nextaxiom.com), which includes a set of software development tools as well as a server infrastructure, has resulted in remarkable improvements in system quality, performance, flexibility, customizability, ease of distribution, and cost and time of implementation as compared to what can be expected from the traditional means of constructing software systems.

[Para 38] As depicted in Figure 4, from the perspective of the present invention, a service-oriented application can be generally broken down into three layers: 1) interface, including the user interface related logic, 2) an optional composite application logic layer including assembly and aggregation of software services and the ability to add logic, and 3) a raw material software services layer (these are services that are indivisible from the perspective of the assembly and management system). These layers can be easily integrated through standards-based software service protocols, such as Web services or any proprietary protocol.

[Para 39] The system of the present invention follows the same general architectural pattern. For example, all the software tools provided by this system such as the log analyzer, as depicted in Figure 5a and Figure 5b, are composed of a graphical user interface, an optional composite application logic layer, and a set of raw material software services.

[Para 40] Any system or definition that references itself is circular by nature. For the purpose of the present invention, and to address this circularity, we define a portion of the system, hereon referred to as the Kernel, which breaks such circularity in avoiding infinitely recursive definitional references. Furthermore, we define the Kernel module as the core functional coordinator, software service dispatcher, and composition manager of the system of the present invention.

[Para 41] In one embodiment of the present invention, a service interface metadata management module, including a design-tool and the metadata management logic, is added to support the basic functionality of the system of the present invention. Since the system of the present invention pivots around the definition of software module interfaces, one of the first modules required

for the construction of such a system is an interface metadata management module. As per the main claim of the present invention, this module is implemented through a set of raw material service interfaces, dispatched by the Kernel module of the system of the present invention, and a software GUI tool for the display, definition and modification of the defined interfaces, that communicates with the system of the present invention through a set of software interfaces designed to add, delete, update and get the metadata associated to the interfaces where the system of the present invention is the dispatcher of such services. Here, to further break the circularity, the only service interface definition that must be hard-coded within the Kernel is the service required for getting/fetching the definition of any other service.

[Para 42] Referring to the Figure 6, you can see the interface design tool in the right pane of the GUI interface displaying the interface of the service used by the interface design tool to fetch the content of the left pane, that is the project pane. In the project pane, you can see the name of some of the other service interfaces for metadata management services used by the design-tools and the runtime platform (a.k.a. server) of the present invention to accommodate the end-user in designing any interface service definition for any software application.

[Para 43] In another embodiment of the present invention, a search tool, that provides granular service discovery functionality to the end-user of the system of the present invention, communicates with the system through a set of software interfaces, as depicted in Figure 7a. Referring to Figure 7b, you can see one of these services implemented as a composite software service, through the system's composition tool. The composite service is implemented by the Kernel and in-turn is assembled from a set of atomic services that are dispatched through the Kernel. Here, once again, we have demonstrated a functionality of the system of the present invention itself constructed through the same functionality used to provide functions to the end-users of the system.

[Para 44] In yet another embodiment of the present invention, the system of the present invention uses a set of service interfaces, as depicted in Figure 8,

to accommodate its service logging functionality. As an example, the Kernel invokes a log-related composite service to log the information about the invocation of any service. The composite service interface and definition are built through the composition and interface design tools of the system of the present invention. Inside the composite service, a raw material service for logging is embedded. At runtime, when a service is invoked through the system of the present invention, the Kernel may attempt to log the invocation of the service by invoking the composite log service which in-turn is executed by the Kernel. The user of the system of the present invention can customize the logging behavior of the system by modifying the composite definition of the logging service. This is an example of how the system of the present invention can be customized through its own end-user functionality.

[Para 45] In another embodiment of the present invention, the system of the present invention uses a set of service interfaces, as depicted in Figure 9a, to accommodate its service monitoring and management functionality. Here, the service management tool, as depicted in Figure 9b, invokes a service on the Kernel to gather information related to the currently running services or to change the attributes, such as priority, of the running services. The Kernel in-turn uses a trap mechanism, that traps named services based on the name associated to the implementation of the service to provide direct implementation for those services (in this case, the Kernel is both the dispatcher and the implementer of an atomic service).

[Para 46] In another embodiment of the present invention, the system of the present invention uses a set of software service interfaces, as depicted in Figure 10a, to accommodate its service cache management functionality. The cache manager GUI, as depicted in Figure 10b, communicates with the system through a set of service interfaces to monitor the cache, retrieve information about the currently cached service, selectively purge cached services, and perform other cache management functions. Upon the invocation of services corresponding to the related software interfaces by the cache management GUI, the Kernel dispatches the implementation of these services.

[Para 47] In yet another embodiment of the present invention, the system of the present invention uses a set of service interfaces to synchronize, persist, define and manage shared memory structures provided to the end-user of the system for inter-service communication. A panel used for the definition and management of a shared memory structure uses a set of service interfaces, dispatched by the Kernel at runtime, to store the definition of the shared memory together with its access control properties. As another example, a different set of services is used by the runtime system to synchronize the modifications performed to one instance of the system of the present invention with the in-memory image of another instance of the system of the present invention.

[Para 48] Other embodiments of this invention, include, but are not limited to: security management, provisioning, cluster management and other software products required by the system of the present invention for providing service interface and service module management, assembly and dispatch functionality.

[Para 49] Upon reference to the description of this invention it is apparent to one skilled in the art that any software product, management and design tool that needs to interact with the system of the present invention can use a set of software service interfaces to interact with the system and that the definition and implementation of those interfaces can be an artifact of the system that is dispatched (as atomic services) or implemented as composite services through the very same functions provided by the system to its end-users.

[Para 50] A further aspect of the present invention is in a layer of encapsulation that provides for the transparent distribution of service invocations. All the services in the system of the present invention are consumed through this layer of abstraction and are hereon referred to as the Invoker interface. Referring to Figure 11, in one embodiment of the present invention, the Hyperservice Business Platform, the software service management and development tools communicate with the required runtime server through an Invoker abstraction. The "InternalInvoker" accesses the runtime environment in the same address space as the consumer of a service.

The "RemoteInvoker" transparently goes through a network protocol layer to access the required runtime environment outside the address space of the consumer of a service. The "RemoteInvoker" may go through a standard protocol such as SOAP/HTTP or any other protocol such as a proprietary protocol. As an example for using the Invoker abstraction, the service log analyzer tool invokes (or consumes) all of the services used for communicating with the system of the present invention through an object implementing the Invoker interface. Two implementations of the Invoker interface are used within the system of this invention; 1) the first is referred to as the InternalInvoker that communicates with the Kernel module within the same computer address space (in other words, the InternalInvoker is in-memory with the Kernel module); 2) the second implementation of the Invoker, hereon referred to as the RemoteInvoker, assumes that the Kernel module of the system of this invention is outside of memory, in other words, it is not within the same computer address space. The RemoteInvoker serializes the invocation request for a software service, usually as, but not limited to, SOAP/XML, and communicates the serialized request usually over, but not limited to, the HTTP protocol. Then it receives the serialized response, including the outputs of the service, and de-serializes it back to the native object form of the requesting service. All operations handled by the Invokers are completely encapsulated from the user of the Invoker interface. In this way, the tools connected to the system of the present invention, can switch between offline InternalInvoker and different servers/runtime instances of the system of the current invention through an array of objects implementing the Invoker interface. Note that each RemoteInvoker instance can be uniquely identified by a separate URL.

[Para 51] Furthermore, the Kernel for the system of the present invention uses the Invoker abstraction to consume all of the system services required for implementing its own functionality such as metadata access, logging services, and others. This has profound implications in the distribution of this system of the present invention. For example, this implies that the Kernel of one instance of the system can access the metadata of another remote instance of

the system by simply using a corresponding RemoteInvoker instead of its own InternalInvoker when consuming metadata data access service interfaces.

[Para 52] In another aspect, the present invention accommodates the service-oriented broadcast of messages and events across all the distributed instances of the system of the present invention. Here, each instance of the system that is aware of the address (URL) of other instances can create RemoteInvokers and broadcast any of the system services to all other instances of the system by re-invoking the same system through each RemoteInvoker. Furthermore, the two-way, asynchronous, service-oriented conversation of any two instances of the system can be accommodated through the addition of a callback invoker address (and security information) to each invocation. In this way instance 1 of the system can invoke a service on instance 2, while providing its own address as the callback address to instance 2. At a later time, instance 2 can invoke a service on instance 1 using the callback invoker address (and security information). One application of this method is to load-balance invocation requests for services without the need to replicate the service interface/composite service definition metadata across two distributed instances of the present invention. Here, instance 1 of the system delegates the consumption of a service to instance 2 of the system and provides an invoker callback address to instance 2 of the system. Instance 2 of the system uses the callback invoker address to instantiate an invoker to be used as the metadata manager. In this way, all the metadata requests for fulfilling the original request given to instance 2, by instance 1, are invoked through instance 1 of the system. Thus, using this approach there is no need for instance 2 of the system to have a duplicate of the metadata for invoking the service whose invocation was requested by instance1.

[Para 53] The specific software tools and systems are representative examples only. Additional software tools and products may be included in the system of the present invention following the same method of dispatch, composition and communication between the software product and the system. The figures depicted herein are representative examples only. There may be many

variations to these figures without departing from the essence of the invention.

[Para 54] Although the present invention has been described with reference to specific embodiments, this description is not to be construed in a limiting sense. Various modifications of the disclosed embodiment as well as alternative embodiments of the invention will become apparent to one skilled in the art upon reference to the description of the invention.